Towards Efficient Query Processing on Massive Time-Evolving Graphs

Arash Fard Amir Abdolrashidi Lakshmish Ramaswamy John A. Miller
Department of Computer Science
The University of Georgia
Athens, Georgia
Email: {ar, ara, laks, jam}@cs.uga.edu

***Abstract*—Time evolving graph (TEG) is increasingly being used as a paradigm for modeling and analyzing dynamic relationships in many emerging domains such as online social networks, World Wide Web and evolutionary genomics. A time-evolving graph consists of a sequence of snapshots of the graph as it evolves over time. The ability to scalably process various types of queries on massive TEGs is central to building powerful analytic applications for these domains. Unfortunately, indexing techniques and cluster computing schemes that have been designed for static graphs are not very effective for processing massive TEGs. Towards designing scalable mechanisms for answering TEG queries, this paper studies three important problems. The first is the distribution of TEG data on the nodes of a cluster computing framework such as Pregel or Giraph so that the computing and communication resources of the cluster are effectively harnessed. The second is the answering of reachability queries on any snapshot of a TEG and the third is that of processing pattern matching queries in TEGs. For each problem, we provide a brief literature survey and explain why trivial extensions of static graph techniques are not adequate for TEGs. We also present our preliminary ideas towards addressing these problems and discuss their benefits.**

Keywords*—time-evolving graphs; big-data; partitioning; reachability; pattern matching

I. Introduction

The importance of graph as a fundamental structure for representing, understanding, and analyzing relationships in many diverse domains can hardly be overemphasized. This is evidenced by the fact that graph storage, querying, computation and mining have continued to be highly active areas of research over the past several decades [1], [2], [3], [4], [5], [6]. Recently, driven by the needs of applications such as social networks, there is a renewed interest in developing scalable frameworks and algorithms for processing massive graphs. Pregel [7], Giraph [8], and GPS [9] are some examples of graph-processing frameworks. In addition to these frameworks, several indexing schemes have also been proposed for scalably and efficiently answering various types of queries on massive graphs. These schemes attempt to mitigate the overheads imposed by on-demand traversal of graphs by typically maintaining certain indexing information on a relational database.

While many of the frameworks and algorithms have been designed and optimized for static graphs (graphs whose node and edges do not change), in many emerging applications, the relationships are not static – they change or evolve over time. Hyperlink structure of the World Wide Web, relationship structures in online social networks, connectivity structures of the Internet and overlays, communication flow networks among individuals, evolutionary history of genome families are some examples of temporally evolving relationships. Recently, *time-evolving graph sequence* (*time-evolving graph* or *TEG*, for short) has been proposed as a modeling and representation paradigm for such dynamic relationship structures [10], [11], [12]. A time-evolving graph is essentially a sequence of *snapshots* of the graph as it evolves over time.

As the numbers and scope of TEG-based applications grow, varieties of TEG computations and queries are increasingly becoming important. Example applications include analyzing the flow of sensitive information in communication flow networks, analyzing duration and stability of relationships and influence among users of social networks, and timely monitoring, management, and root cause analysis of large-scale overlays and distributed systems.

The current graph computation frameworks and query mechanisms are not adequate for TEGs because of the additional challenges they pose. First, in comparison with static graphs, TEGs involve an additional dimension, namely *time*. This additional dimension manifests itself in multiple temporally distinct classes of queries such as historical (or snapshot-specific), inverse-temporal and continuous queries. Accommodating this additional dimension into indexing and query-processing algorithms is a significant challenge. Second, TEGs in many modern domains are huge. For example, Facebook Friendship graph is estimated to have around 800 million vertices and 104 billion edges [13]. Furthermore, the additional temporal dimension of TEGs causes the data size to increase by multiple orders of magnitude. Dealing with data of this magnitude demands extreme scalability and efficiency. Obviously, we will need a cluster-based infrastructure for processing massive TEGs. In these clusters, the TEG data is distributed among multiple machines. The manner in which data and computation tasks are distributed among the machines has significant impact on the performance of the system. Since the structure of a TEG changes over time, it is not only necessary to design effective strategies for distributing indexing and querying tasks among multiple machines, but also devise incremental schemes for maintaining the distributions over time taking into account the graph dynamics.

COLLABORATECOM 2012, October 14-17, Pittsburgh, United States
Copyright © 2012 ICST
DOI 10.4108/icst.collaboratecom.2012.250532

In this paper, we explore the above challenges in the context of three important time-evolving graph problems. Our first problem is that of distributing or partitioning dynamic graph data on the set nodes of a shared nothing compute cluster and maintaining the distributions as the graph evolves over time. The distribution should optimize the overall performance of the cluster both in terms of computation and communication. Our second problem is to efficiently answer reachability queries in dynamic graphs. Reachability query tests whether there is at least one path between a given pair of source and destination vertices in a particular snapshot of a time evolving graph. The third problem we study is that of answering subgraph queries in a large dynamic graph. This type of query finds subgraph instances within a given snapshot of a dynamic graph that match to a query graph. Both reachability queries and subgraph queries have wide applications in diverse areas such as XML query processing and bio-informatics. While both of these queries have been well studied for static graphs, there is very little work on supporting these queries for dynamic graphs. For each of the three problems, we provide a formulation, we discuss why existing techniques from static graphs cannot be trivially extended for dynamic graphs, and we provide possible approaches for addressing the problem.

The rest of the paper is organized as follows. In section II we examine challenges of partitioning TEGs among the nodes of a cluster. Section III is an investigation about running reachability queries in TEGs. We also explore the issues of pattern matching for TEGs in section IV. Each of these sections introduces some related concepts and formulates the problem; then reviews related works, and finally suggests our approach. The last section is dedicated to conclusion of the paper.

II. TEG Distribution on Clusters

This section first shows the importance of graph partitioning on the efficiency of any distributed graph processing algorithm; and then, summarizes the available approaches. We also explore the pros and cons of each approach. We specifically consider the factors that affect the result of the partitioning of a time evolving graph.

A. Problem formulation

Workload distribution - the distribution of data among a set of processors - has been one of the fundamental issues in any parallel and distributed processing applications. Most of the data sets in these applications are not regular in structure and can be presented in a graph-based form. Each node of such a graph represents a unit of data where some form of computation will take place while each edge represent a dependency between these units of data. Undoubtedly, the way one partitions such a graph and distributes the data nodes among processor nodes of a cluster will affect the performance of computation. The performance of the system can be measured with respect to two metrics, namely cost of communication among compute nodes in the cluster and utilization of nodes in terms of the time that compute nodes are not idle. An optimal solution would minimize the overall runtime of the application by satisfying two contradictory goals, minimizing the communication cost among processors and maximizing the node utilization. Before explaining why the two metrics may compete against each other, we describe shortly the model of computation for performing graph algorithms on a cluster of computers.

B. Bulk Synchronous Parallel

It has been show in [7], [14] that systems based on MapReduce [15] are not always suitable when processing large graphs. This is mainly because writing graph algorithms in form of map and reduce functions are not efficient because the state of the graph must get written to and read from HDFS constantly during each map and reduce job which are costly operations.

Bulk Synchronous Parallel (BSP) [16] introduced by L. Valiant in the 90's is a model of computation for parallel processing where processors are connected to each other to form a distributed shared nothing cluster. Each of these computers has their own thread of computation. Computation in BSP consists of iterations called supersteps. Each super step consists of three ordered stages: Concurrent computation where each active computer does its computation based on its local data and asynchronous form other nodes, Communication where all of the computers send each other their messages and Barrier Synchronization in which finished processes wait for other unfinished processes to finish their communication so they become synchronous.

Using the BSP model of computation for performing an algorithm on a graph, one way to maximize the compute node utilization - hence decreasing the time that they are idle - is to randomly assign the vertices of the graph to different compute nodes. This way of assignment increases the node utilization because upon completion of local computation on a vertex, the vertex will have to send messages to its adjacent nodes residing in different compute nodes (because of random assignment of vertices to nodes). However, this approach increases the amount of messages need to be send among the compute nodes resulting an increased communication cost.

On the other hand, in order to minimize the communication cost one way is to partition the graph to a number of highly connected subgraphs with few edges among them and distribute each of these partitions among different compute nodes. However, this approach can lead to high idle time of compute nodes hence wasting resources and decreasing the node utilization. This is due to the fact that the computation of the algorithm (based on BSP model of computation) starting from an initial partition can take long time to propagate throughout all of vertices of the subgraphs residing in other partitions on other nodes because of the highly connected structure of the initial subgraph as depicted in figure 1. Based on these two extremes, we propose ways and mechanisms that distributed graph-processing systems can utilize to improve the performance.

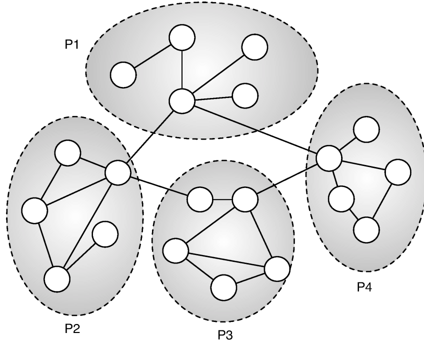

Fig. 1. Assignment of graph vertices to four processor based on a graph partitioning algorithm

C. Related Work

Pregel, Apache Giraph, Mizan [17] and GPS are some of the BSP inspired systems. Pregel and Giraph distribute vertices of a graph among the compute nodes randomly (in round robin fashion). However, none of them consider the effects of graph partitioning algorithms on the computation of graph-based algorithms. Mizan considers an extra layer between the graph algorithm API and the computation infrastructure. It can either perform a min cut on the graph to partition it to many as compute nodes or use random assignment of nodes to partition the graph. This approach can suffer from expensive computation times. Moreover, it does not consider the node utilization of compute node, nor does it consider the TEGs.

GPS is similar to other systems but has three main features. It has an extend API to support graph algorithms that are both vertex centric and global centric. It also considers dynamic repartitioning of the graph based on messages to balance the workload. Moreover it utilizes an optimization method to distribute high degree vertices across compute nodes. However, it does not consider the graphs that are time evolving and how this affects the partitioning mechanisms.

D. Our Approach

As mentioned earlier both assigning the vertices randomly and partitioning the graph into connected components with minimum edges among them have major benefits. Hence, a balanced trade-off between these two extremes will benefit any distributed graph processing system. One way to accomplish this is to consider what factors affect the result of the partitioning of a time evolving graph. We believe three aspects should be considered: the computation being performed, the dynamic structure of the graph and the way the compute nodes are configured to perform the computation. Below we explain each of them.

The main goal of partitioning is to reduce the number of inter-partition messages. Hence, algorithms that are communication intensive and have low computation cost at each vertex will benefit most. Example of such algorithms are PageRank [18] and Highly Connected Components (HCC) as implemented in [19]. Whereas algorithms such as Single Source Shortest Path (SSSP) which have low cost of communication comparing to PageRank may not benefit from graph partitioning. The performance of such algorithms can be improved by increasing the node utilization if the assignment of vertices to nodes is done in a random fashion. It is our contention that the overheard of partitioning a graph to highly connected subgraphs is not beneficial for computation such SSSP

The dynamicity of the TEGs is through addition/deletion of nodes or edges or both. This can take place within a partition residing on a compute node or among partitions. Dynamic repartitioning of subgraphs seems to be the logical response. However, this should be done carefully because dynamic repartitioning can be expensive, since the assignment of vertices to compute nodes needs to be changed. The cost is due to the fact that the graph processing system must make sure that before communication phase of a BSP super-step starts, all vertices are aware of the possible new assignments of their adjacent vertices to compute nodes so they can send their messages to. This necessitates a repartitioning that is beneficial. In other words, repartitioning should be performed when addition/deletion of nodes or edges pass a certain threshold related to the connectivity and structure of the subgraphs. In this manner, we can be sure that repartitioning takes place when its benefit outweighs the cost.

The other option that can benefit the communication cost is the incremental reallocation of a node in a subgraph to another subgraph (processed by another processor). In this case, when a node in a subgraph starts communicating with another node in another subgraph, we can send a replica of that node to the destination partition. This reduces the network communication cost because instead of sending multiple messages we only need to send one message between partitions to keep the similar copies of the same node.

Finally, in order to improve the performance and maximize the node utilization, the way the compute nodes are configured to perform the task is important. Our approach is to generate more partitions than the available compute nodes. In this manner, by reducing the size of the graph partitions we are making sure that the compute nodes have more lighter workloads to perform rather than a few heavy workloads. This approach will also lower the overall runtime of the algorithms on the system. Since the run time of computation based on BSP is bounded by the slowest worker, by constructing smaller workloads for the compute nodes we are making sure the slowest worker will finish the computation sooner. We are not aware of any distributed graph processing system that considers both of these two metrics while processing graph algorithms on time evolving graph. Our approach can improve the performance of the cluster based graph processing systems like Pregel, GPS and Giraph.

III. Reachability Query in TEGs

Consider a time evolving directed acyclic graph G. Let $\{G_1, G_2, \ldots, G_q, \ldots, G_r\}$ be the different snapshots of the graph. Let Diff(G_q, G_{q-1}) represent the changes occurring

between snapshots G_q and G_{q-1}. Note that the Diff between any two snapshots can be represented as a union of a set of vertex additions, a set of vertex deletions, a set of edge additions and a set of edge deletions.

The reachability query `Reach(v, w, q)` seeks to find out whether node w was reachable from node v in the qth snapshot of the time evolving graph – the answer should be TRUE if w was reachable from v in G_q and FALSE otherwise.

A. *Reachability Analysis in Static Graphs*

There has been considerable interest in efficient answering of reachability queries in static graphs. Several approaches such as transitive closure, on-demand depth-first traversal/breadth-first traversal, interval-based indexing, 2-HOP indexing have been studied in the literature [20], [21], [22], [23], [24]. These various approaches form a spectrum with pre-computation of transitive closure and on-demand graph traversal lying at its two ends [22]. Pre-computing transitive closure has heavy indexing costs while the query-time is constant. On the other hand, on-demand traversal has no indexing costs but its query time is $O(N+M)$, where N and M are respectively the numbers of nodes and edges in the graph.

Interval-based indexing has received considerable research attention because it provides a good balance in the trade-off between indexing costs and query-time. While the various interval-based index schemes share a common paradigm, they do differ considerably from one another. We first discuss the common aspects and then highlight the differences between them. Interval-based indexing techniques start by identifying a spanning tree (although GRAIL [22] uses multiple spanning trees, we limit our discussion to a single spanning tree). This tree is traversed in the depth-first order and each node is assigned a pre-order and post order number. The node-ids and their pre- and post-order values are stored in a table. The node w can be reached from node v using only the edges of the spanning tree if and only if the pre-order value of w is in between the pre and post-order values of v (i.e., $v_{\text{pre}} < w_{\text{pre}} < v_{\text{post}}$).

The above pre- and post-order index-based querying has to be augmented in some fashion to account for reachability offered by paths that include at least one non-tree edge. The individual interval-based indexing schemes differ in how they analyze such paths. For this purpose, the DualLabeling technique [21] maintains a transitive link table (TLT) that contains the transitive closure of the non-tree edges. The paper presents a sophisticated algorithm that utilizes the TLT and the pre- and post-order index to answer reachability queries in constant time.

The GRIPP indexing [20] on the other hand, does not pre-compute the transitive closure of non-tree edges. It expands the table containing pre- and post-order indexes to also store non-tree edges. Every node in the tree gets a pair of pre and post order index values reflecting its position in the spanning tree. However, if a node, say y has one or more incoming non-tree edges, it receives an additional pair of values corresponding to each incoming non-tree edge. Specifically, suppose (v, y) is a non-tree edge. During the spanning-tree traversal, when node v is reached, the link (v, y) is traversed as though it is a tree edge and node y is assigned a pre-order value. However, this non-tree instance of y is treated as a leaf node (although y may have outgoing edges), and it is immediately assigned a post-order value as well. This pair of values is stored in the pre- and post-order index table, but specifically marked as 'non-tree'. Reachability testing is done by computing multiple *reachability instance sets* through recursive containment queries. The reachability testing in GRIPP is $O(M-N)$, where M and N are the numbers of edges and nodes in the graph respectively.

B. *Limitations of Existing Approaches*

A naive way of applying the interval-based indexing strategy for answering ss-reachability queries is to index each snapshot of the graph using any of the current techniques. The ss-reachability `SSReach(v, w, q)` can then be answered by using the index values corresponding to the qth snapshot. Figure 2 shows three snapshots of a time-evolving graph containing 8 nodes A through H (note that the node r is a fictitious root node that is added by GRIPP to deal with certain special conditions). The figure also provides three tables containing the pre- and post-order indexes corresponding to the three snapshots. In the tables 'Tr' and 'Nt' indicate index values obtained by traversing tree and non-tree edges respectively. Now, if we want to answer `Reach(E, H, 3)`, we use the rightmost table and recursively compute reachability instance set of E and B. Since H is in the reachability instance set of B, the answer is TRUE.

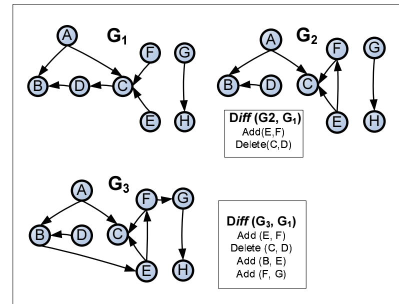

Fig. 2. Reachability Query Processing in TEGs

However, there are many drawbacks to this simple approach. First, the computational overhead of indexing every snapshot is going to be very high, as it will require traversal of each snapshot. Second, the storage overheads are going to be high as well because of the need to store the index of every snapshot. Both the computational and storage costs are exacerbated as the hierarchies increase in size and as they change more frequently. Third, there may be very few queries on some certain fraction of the snapshots in which case indexing every version is wasteful both in terms of storage and computation. However, it should also be noted that query distribution in terms of snapshots is not known apriori. Fourth, for large and frequently changing hierarchies, the huge amount

of indexing data causes the database to become bulky which significantly increases query latencies.

C. Our Approach

Our approach for answering reachability queries in TEGs is to index only a selective interspersed subset of snapshots (using interval-based indexing technique). For example, in Figure 2, only snapshots G_1 is indexed. For the rest of the snapshots (e.g., G_2 and G_3 in Figure 2, only the changes from the previous snapshot is stored in a Diff List or an Edit List (shown below each snapshot).

The issue however is *how to answer reachability queries on snapshots that are not indexed* (e.g., in Figure 2 is H reachable from A in G_3)? In our approach, the query `Reach(v, w,` G_q`)`, where G_q is non-indexed will be answered in two steps – we answer the query `Reach(v, w,` G_p`)` (G_p being the temporally closest indexed snapshot to G_q [1] and then checking whether the changes occurring between G_p and G_q alters the reachability status. In other words, if w was reachable from v in G_p, we need to figure out whether the changes in Diff(G_q, G_p) make it unreachable, and vice-versa. In the TEG shown in Figure 2 answering the query `Reach(A, D,` G_3`)` will require us to figure out whether the changes in Diff(G_3, G_1) breaks the reachability (since `Reach(A, D,` G_1`)` is TRUE). The straightforward approach of processing *all* the changes in Diff(G_q, G_p) in chronological order and reflecting each update's effect on G_p will turn out to be very inefficient because most of the changes (in some cases all of the changes) will have no impact on the reachability status.

Thus, our approach is to find out which changes in the Diff are likely to impact the reachability status and process only them. The central question then is *how do we correctly figure out the changes in Diff(G_q, G_p) that will impact the reachability of w from v?* Changes that seem unrelated at first glance might in fact have an effect on the reachability because of other chronologically subsequent changes. In our example, suppose we are processing the query `Reach(A, H,` G_3`)`. The change `Add E, F` might seem unrelated to the query. However, this change along with `Add B, E` and `Add F, G` alters the reachability from A to H between snapshots G_1 and G_3. In our preliminary work, we have made two key observations that will help us address this challenge. First, suppose `Reach(v, w,` G_p`)` evaluates to be TRUE, Diff(G_q, G_p) could possibly alter the reachability status if it contains at least one edge deletion – `Delete(u, y)` where `u` and `y` are two consecutive nodes in a directed path from v to w in G_p. Second, suppose `Reach(v, w,` G_p`)` evaluates to be FALSE, Diff(G_q, G_p) could possibly alter the reachability status if it contains at least one edge addition – `Add(u, y)` where either u is a node that is reachable from v in G'_q or y is a node from which w is reachable in G_p. Note that both of these are necessary, but not sufficient conditions. Our approach will leverage these observations in the following broad manner.

[1] G_p may temporally precede or succeed G_q. For simplicity, in this paper we assume that G_p precedes G_q.

Rather than processing all changes in chronological order, we *search* Diff(G_q, G_p) for a change that can possibly alter the current reachability status, and if such an update is found we process it, and reflect its effect on the reachability status. If no such changes are found the process terminates. In our example (Figure 2), when answering the `Reach(A, H,` G_3`)`, we first process the update `Add B, E` (since B is reachable from A). Subsequently, we process the updates `Add E, F` and `Add F, G`, to conclude that `Reach(A, H,` G_3`)` is TRUE.

Our approach has several advantages. First, since all snapshots need not be indexed, it provides significant reduction in index computation costs as well as index storage costs. Second, it also improves the query latencies for very large TEGs. This may seem counter-intuitive (one would expect higher query latencies when lesser number of snapshots are indexed). However, it is because the database is less bulky when only a subset of snapshots are indexed which leads to better query performance. Third, our approach provides flexibility to the application in deciding the indexing costs that it can tolerate.

IV. TEG Pattern Matching

Graph pattern matching is a fundamental problem in graph processing and can encompass a number of different problems, ranging from subgraph isomorphism which is very restricted and NP-complete [25] until graph simulation which is more relaxed and can be determined in quadratic time [26]. Basically, it is finding all the matches of a given graph, called query graph, in an existent larger graph, called data graph. To define it more formally, assume that there is a data graph $G(V, E, l)$, where V is the set of vertices, E is the set of edges, and l is the set of the labels of the vertices. In general, edges can be directed or undirected, but here we assume the edges are directed and the vertices are labeled unless it is mentioned explicitly. There is also a pattern or query graph $Q(V_q, E_q, l_q)$ which forms the interesting pattern that we want to find its occurrences in the data graph. The task is finding all subgraphs of G that match the query Q. By definition, $G'(V', E', l')$ is a subgraph of G if and only if V' is a subset of V and E' is a subset of E.

Matching can be defined by structural or semantic matching and its related algorithms can be designed for exact or approximate solutions. Moreover, these algorithms can be designed and executed in sequential or parallel, centralized or distributed fashion. Choice of matching can dramatically affect efficiency and complexity of the problem, and in this section we first introduce some different types of matching. Then, we review some of the recent work in incremental graph pattern matching and distributed graph pattern matching. Regarding the fact that it is fairly a new research topic, we found only few numbers of related papers, and to the best of our knowledge there is no work done to integrate incremental and distributed pattern matching algorithms on large data graphs. At the end of this section, we provide a formulation to address this problem.

A. Different Graph Pattern Matching Paradigms

Each of these paradigms may suite to a different type of applications. They are all well defined in the literature, and here we just summarize some of them. Subgraph isomorphism is more restricted and preserves topological structure of the query, while graph simulation takes care of child relationships between different types. Graph simulation is studied in the recent years because its applications in analysis of social networks.

1) Subgraph isomorphism: It is a bijective mapping between a query graph $Q(V_q, E_q)$ and a subgraph of a data graph $G(V, E)$. That is, assuming $G'(V', E')$ as a subgraph of G, graph Q will be subgraph isomorphic of G if there is a function $f : V' \rightarrow V_q$ and for any v' and w' belonging to V' there are v_q, w_q belonging to V_q such that edge (v', w') belongs to E' if and only if edge (v_q, w_q) also belongs to E_q.

2) Simulation and its extensions: Graph simulation is less restricted than traditional matching paradigms like isomorphism; hence, asymptotically faster. It plays an important role for recently emerging applications of analyzing social networks. It is said that graph $G(V, E)$ matches a pattern $Q(V_q, E_q)$ via graph simulation if there is a binary relation $R \subseteq V_q \times V$ such that if $(u, u') \in R$, u and u' have the same label; moreover, for every $u \in V_q$ there is a u' such that $(u, u') \in R$, and also for every $v \in V_q$ that $(u, v) \in E_q$ there is a $v' \in V$ such that $(v, v') \in R$ and $(u', v') \in E$. With respect to usefulness of graph simulation in analyzing new applications of pattern matching, a few extensions have been introduced during recent years such as bounded simulation [27], strong simulation, and dual simulation [28].

B. Related Work

In many web applications data are generated and stored in distributed manner. Even when web-scale data are stored in a single location, they are so large that they are distributed among machines of a cluster system in a data-warehouse. In order to efficiently process the huge graphs constructed from these data, a few distributed graph processing systems are developed during the last years like Pregel, Trinity [29], and GPS. Although providing an efficient distributed graph pattern matching can be very useful for extracting information out of big data, it is not very well explored. In fact, because of the huge size of the graph, the old approaches are not applicable; for example, exhaustive indexing would be infeasible. Moreover, poor locality of real data graphs causes inefficiency in distributed processing. In the following, we briefly explain recently suggested approaches in [30] and [31] about distributed graph pattern matching.

On the other hand, it is important to have incremental pattern matching in order to avoid running the same procedure for the whole data graph when it is a TEG. Considering the fact that it is relatively a new emerging field of research, there are only few papers on incremental pattern matching. [32] has explored some aspects of this problem; however, it does not cover distributed incremental pattern matching.

In [30] a distributed algorithm is introduced for finding isomorphic subgraph matches of a given query graph in a huge data graph. They deploy graphs on Trinity which provides a distributed memory cloud environment. In their system any given query graph is first decomposed by the master node into a few two-level tree structures, called STwig. Then, these sub-queries, which are set in an order to minimize the size of intermediate results, will become available to all computing nodes as a query plan. The matching results for these sub-queries are found via graph exploration in parallel on partitions of a data graph locally available in the memory of each node; these results should be joined to each other to find the results for the complete graph query. The computing nodes also need to exchange some of their intermediate results in order to join them and create the final results of the query. This work focuses on efficient web-scale graph processing, so it avoids exhaustive usage of indices which is very common in most of graph processing systems. Indeed, they only use a simple string index which maps node labels to node IDs. It is argued that the time needed for constructing indices and the capacity for their storage would be infeasible in web-scale graph processing. Index size and time of the proposed approach, in addition to the query processing time, is compared with estimation for some other existent systems. However, it seems that the proposed system is only compared with non-distributed systems, and other possible solutions for efficient distributed processing, such as distributed indexing, are not explored. On the other hand, Trinity is built on top of a distributed memory storage layer which provides a transparent interface for users to work with a distributed graph as if it was stored in memory of a single machine. The subgraph matching approach which is proposed in the paper is experimented on a cluster of at most 12 machines, but it is not clear whether it is scalable for a larger number of machines or not, considering the fact that providing a transparent memory space is not very scalable inherently.

In [31] a distributed algorithm for graph simulation is proposed. They have also analyzed distributed algorithms for graph simulation and identified three complexity measures for their analysis: (1) visit time, which measures maximum visiting time of a machine in a cluster system and indicates the complexity of interactions, (2) makespan, which is response time to the query from submitting query until when its answer is ready, (3) data shipment, which is the total size of messages exchanged between machines of the cluster system during computation. They have implemented and tested their algorithm on a cluster of 16 machines; however, neither the algorithm nor its implementation is adapted to a widely in use cluster computing platform like Pregel, but it is implemented all in Python.

The study presented in [32] investigates incremental algorithms for simulation, bounded simulation, and subgraph isomorphism. It specially analyzes whether the problems are bounded or not. By the definition given in that paper, an incremental problem is said to be bounded if the cost of its update is a function of the size of the changes in input and

output. For incremental pattern matching, the authors keep a set of result graphs representing the found matches in the data graph. Then they adjust this result graph to update it based on the changes in data graph.

C. Our Approach

We suggest an approach for efficiently processing distributed subgraph matching on BSP framework. We also integrate a mechanism to support incremental processing which we believe can dramatically improve the response time of graph queries on a distributed dynamic graph. For implementation we use GPS which its architecture is very similar to Pregel. It means that vertex-centric algorithms can be implemented easily; in addition, it has an extension to enable efficient implementation of algorithms composed of more than one vertex. In Pregel-like systems, each vertex is aware of its own label and its outgoing edges.

Assuming that a large directed data graph $G(V, E, l)$ is distributed among many workers according to a particular partitioning paradigm, we use GPS features and graph exploration technique to efficiently find maximum graph simulation match in G for a given Query Q. The system contains one master process and many worker processes. Each worker is responsible for a partition of G which may consist of many vertices. In general, a processing unit is dedicated to each worker. The input of an appropriate algorithm would be a directed graph $Q(V_q, E_q, l_q)$ as pattern (query) graph, and its output would be the maximum match M in G for Q.

Following we explain two algorithms, the first one is a purely vertex-centric algorithm which would be easier to code and enjoys high level of parallelism; however, it may not be very efficient when the number of vertices is large. The second one is at worker-centric and enjoys a distributed index for finding vertices given a label in each worker. In more detail, the distributed algorithms work as it follows:

1) Vertex-centric: At the first superstep, the master node of the system after receiving a query graph will broadcast it to all the workers. At the second superstep, any vertex of G which has the same label in Q will flag its membership in a set of match nodes, called S. Vertices in S need to know at least one list of the labels for their children dictated by their potential match in Q. It is possible to have two vertices in Q with the same label, so a vertex of G might match to more than one vertex of Q; moreover, the correspondent vertex in Q may not have any child. Hence, members of S may need to remember more than one list or just one empty list as the legitimate labels of their children. Obviously, if the number of outgoing edges in a vertex is less than any list of its children, the vertex should remove itself from S. At the end of this superstep, any member of S sends its ID to its all children.

In the next superstep, vertices which have received a message save the IDs of their parents, and reply back with their own label, ID, and the membership status in S. The vertices receiving message in the next superstep are member of S and can figure out if the label of their children in data graph, returned by them in the messages, is a superset of the labels of their match's children in the query graph. Those that cannot satisfy this condition should leave S. Any vertex which removes itself from S, will report it to all of its saved parents. When a vertex receives a removal message from a child, first the child will be removed from its list of saved children, and then will be examined again to make sure it still satisfies the condition of being in S. If the remained children do not satisfy the required ones, the vertex should remove itself from S and report it to its all saved parents. This will be repeated until S is purified to only the true graph simulation matches. When there is no more communication among the nodes which means they all voted to halt, the master will broadcast to all the nodes and asks about the members of S.

2) Worker-centric: For this algorithm, the query is assumed to be a rooted directed graph. This assumption does not make lose of generality because it can be proved that each directed graph can be decomposed to a few rooted directed graphs in such a way that the union of their graph simulation results is equal to the result of the initial graph. First, the master broadcasts the query to all workers. Each worker will find the set of potential matches for the query's root vertex among the vertices of G which are located at that worker using a local index. Then, the children of these vertices will be explored for having the appropriate set of labels. It continues to traverse the subgraphs while consequent matches vertices are found and wrong subgraph matches are removed.

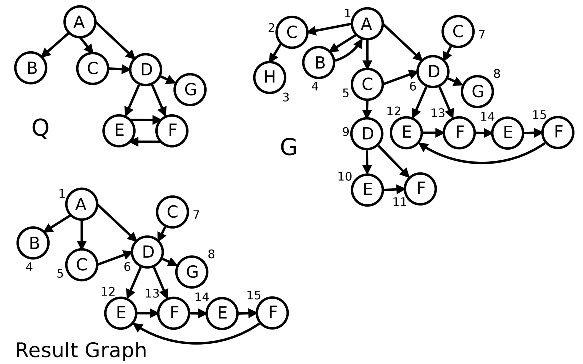

Fig. 3. A sample data graph G, query Q, and Result Graph

In the aforementioned algorithms, vertices located at the same partition will be controlled by the same worker, and there would be no real network communication among them. The edges between vertices on different workers do not hurt the accuracy of the algorithms, but decrease the performance; hence, an appropriate partitioning of G, using methods suggested in section II, will be useful to improve efficiency of the system. Moreover, to address graph simulation matching in TEGs we borrow the idea of *result graphs* from [32] to store the answer of different snapshots with their appropriate time stamps in order to evolve between them. We make separate lists for requests of insert and delete, and add time stamps to each entry. Time stamps help to recognize which changes in the lists of *Insert* and *Delete* should be considered for evolving the result graph of one snapshot of G to the next snapshot. It

is obvious that deleting an edge from G can only diminish the result if it already exists in the previous result graph. Adding a new edge to G will be also considered with respect to expansion of previous result graph. A sample data graph G, a sample query Q, and the result graph of the simulation matching in G for Q is illustrated in figure 3.

V. Conclusion

Research activity on processing very large data graphs on cluster systems has increased lately. Common classes of problems include graph partitioning, reachability, and subgraph pattern matching. As a form of big data, processing massive data graphs requires innovation in the development of distributed algorithms to run on high-performance clusters. While innovation is needed for both distributed query processing and graph updates, particularly for massive time-evolving graphs, there is little work done in this area. This paper surveys the state-of-the-art and discusses avenues for future research. In our ongoing work, we are going to implement approaches suggested in this paper and evaluate them through experiments.

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *SIGMOD Conference*, 1989.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, 1959.

[3] X. Yan, P. S. Yu, and J. Han, "Substructure similarity search in graph databases," in *SIGMOD Conference*, 2005.

[4] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *SIGMOD Conference*, 2008.

[5] U. Kang, M. McGlohon, L. Akoglu, and C. Faloutsos, "Patterns on the connected components of terabyte-scale graphs," in *ICDM*, 2010.

[6] U. Kang, D. H. Chau, and C. Faloutsos, "Mining large graphs: Algorithms, inference, and discoveries," in *ICDE*, 2011.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing - "abstract"," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, ser. PODC '09. New York, NY, USA: ACM, 2009, pp. 6–6. [Online]. Available: http://doi.acm.org/10.1145/1582716.1582723

[8] "Giraph website," http://giraph.apache.org/.

[9] S. Salihoglu and J. Widom, "Gps: A graph processing system," Stanford University, Technical Report, 2012. [Online]. Available: http://ilpubs.stanford.edu:8090/1039/

[10] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu, "Graphscope: parameter-free mining of large time-evolving graphs," in *KDD*, 2007.

[11] S. Asur, S. Parthasarathy, and D. Ucar, "An event-based framework for characterizing the evolutionary behavior of interaction graphs," in *KDD*, 2007.

[12] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins, "Microscopic evolution of social networks," in *KDD*, 2008.

[13] "Facebook User Statistics," http://www.facebook.com/press/info.php.

[14] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, pp. –1–1, 2010.

[15] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[16] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[17] P. Kalnis, Z. Khayyat, K. Awara, and H. Jamjoom, "Mizan: Optimizing graph mining in large parallel systems." [Online]. Available: http://hdl.handle.net/10754/217609

[18] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998. [Online]. Available: http://dx.doi.org/10.1016/S0169-7552(98)00110-X

[19] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 229–238. [Online]. Available: http://dx.doi.org/10.1109/ICDM.2009.14

[20] S. Trißl and U. Leser, "Fast and practical indexing and querying of very large graphs," in *SIGMOD Conference*, 2007.

[21] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu, "Dual Labeling: Answering Graph Reachability Queries in Constant Time," in *ICDE*, 2006.

[22] H. Yildirim, V. Chaoji, and M. J. Zaki, "GRAIL: Scalable Reachability Index for Large Graphs," *PVLDB*, vol. 3, no. 1, 2010.

[23] R. Jin, L. Liu, B. Ding, and H. Wang, "Distance-Constraint Reachability Computation in Uncertain Graphs," *PVLDB*, vol. 4, no. 9, 2011.

[24] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu, "Fast computing reachability labelings for large graphs with high compression rate," in *EDBT*, 2008.

[25] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[26] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs," in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, ser. FOCS '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 453–. [Online]. Available: http://dl.acm.org/citation.cfm?id=795662.796255

[27] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: from intractable to polynomial time," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 264–275, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920841.1920878

[28] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 310–321, Dec. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2095686.2095690

[29] H. W. B. Shao and Y. Li, "The trinity graph engine," 2012.

[30] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. VLDB Endow.*, vol. 5, no. 9, pp. 788–799, May 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2311906.2311907

[31] S. Ma, Y. Cao, J. Huai, and T. Wo, "Distributed graph pattern matching," in *Proceedings of the 21st international conference on World Wide Web*, ser. WWW '12. New York, NY, USA: ACM, 2012, pp. 949–958. [Online]. Available: http://doi.acm.org/10.1145/2187836.2187963

[32] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu, "Incremental graph pattern matching," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 925–936. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989420